Bit-Level Partial Evaluation of Synchronous Circuits

Sarah Thompson
Computer Laboratory
University of Cambridge
sarah.thompson@cl.cam.ac.uk

Alan Mycroft
Computer Laboratory
University of Cambridge
am@cl.cam.ac.uk

ABSTRACT

Partial evaluation has been known for some time to be very effective when applied to software; in this paper we demonstrate that it can also be usefully applied to hardware. We present a bit-level algorithm that supports the partial evaluation of synchronous digital circuits. Full PE of combinational logic is noted to be equivalent to Boolean minimisation. A loop unrolling technique, supporting both partial and full unrolling, is described. Experimental results are given, showing that partial evaluation of a simple microprocessor against a ROM image is equivalent to compiling the ROM program directly into low level hardware.

1. INTRODUCTION

Partial evaluation [12, 14, 13] is a long-established technique that, when applied to software, is known to be very powerful; apart from its usefulness in automatically creating (usually faster) specialised versions of generic programs, its ability to transform interpreters into compilers is particularly noteworthy.

In this paper, we present a partial evaluation framework for synchronous digital circuits that, whilst supporting specialisation, also supports the first Futamura projection [10]:

$$PE[\![interpreter, program]\!] = compiler[\![program]\!].$$

In hardware terms, this is equivalent to taking the circuit for a processor and a program ROM image, then compiling this into hardware that represents the program only – as with software partial evaluation, the processor itself is optimised away, leaving only the functionality of the program expressed directly in hardware.

Note that in this paper, we consider only the partial evaluation of purely synchronous circuits, i.e. circuits consisting only of acyclic networks of gates, with feedback occurring only via D-type latches whose clock inputs are all driven by a single global clock net. Generalisation to the asynchronous case is discussed briefly in Section 7.1.

In Section 2 we discuss PE of combinational circuits; in Section 3 this is extended to encompass loop unrolling of synchronous circuits. Section 4 introduces HarPE, the hardware description language and partial evaluator used to support the experimental work leading to the results described in Section 4.

ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Charleston, South Carolina, January 9-10 2006

Table 1: Rewrite Rules for Combinational PE

$a \wedge a \rightarrow a$	$a \vee a \rightarrow a$
$a \wedge \mathit{false} \rightarrow \mathit{false}$	$a \wedge \mathit{true} \rightarrow a$
$a \vee \mathit{false} \rightarrow a$	$a \vee \mathit{true} \rightarrow \mathit{true}$
$\neg \mathit{false} \rightarrow \mathit{true}$	$\neg \mathit{true} \rightarrow \mathit{false}$

2. PE OF COMBINATIONIAL CIRCUITS

The simplest form of hardware partial evaluation is already well known to hardware engineers, though under a different name: Boolean optimisation. For example, the combinational circuit represented by the expression

$$a \wedge (b \vee c)$$

where a, b and c are inputs, may be specialised for the case where c is known to be *true* as follows:

$$a \wedge (b \vee \mathit{true}) = a \wedge \mathit{true} = a$$

Boolean optimisation is well-studied, with many approaches documented in the literature. Some techniques are sub-optimal but can be applied to any circuit, whereas others (e.g. OBDDs, flattening to CNF or DNF) can yield optimal results[1] but suffer exponential size blowup when confronted with some circuits, particularly multipliers. Any of these techniques are potentially capable of PE of combinational circuits, though for clarity and generality, we have opted for a simple approach based upon term rewriting. Table 1 shows a simple set of rewrite rules that are sufficient to implement (sub-optimal) combinational PE in time and space that is linear with respect to the original circuit.

The software analogue of a combinational circuit, from the point of view of PE, would be a program consisting only of assignment statements, *if-then* and *if-then-else* constructs, but strictly no loops.

[1]In hardware terms, 'optimal' is not easily defined. In some cases, circuits are optimised for minimum gate count, though more commonly they are optimised for speed or power consumption – only rarely will a circuit be optimal with respect to more than one of these considerations.

Figure 1: General Form of Synchronous Circuits

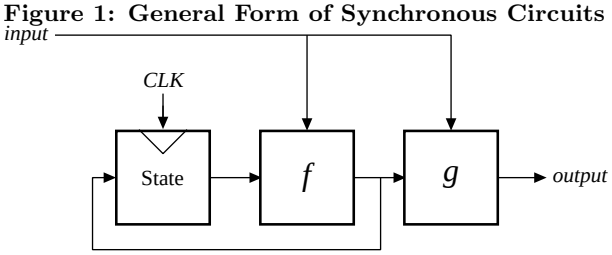
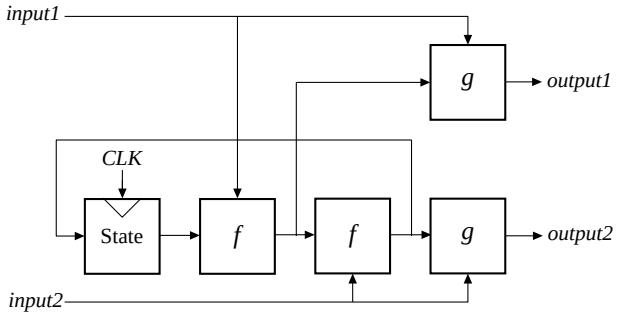


Figure 2: Synchronous Circuit After One Unrolling



3. PE OF SYNCHRONOUS CIRCUITS

A slight variation[2] on the general form of any synchronous circuit, generally referred to as a Mealy machine [17], is shown in Fig. 1. In software terms, such circuits resemble a program of the form

```
while true
    state_{k+1} := f(state_k, input)
    output := g(state_{k+1}, input)
endwhile
```

Each iteration of the loop represents exactly one clock cycle. The internal state of the circuit, represented by $state_k$, may change only once per clock cycle and is determined only by the result of the combinational Boolean function $f : (\mathbb{B} \times \cdots \times \mathbb{B}) \times (\mathbb{B} \times \cdots \times \mathbb{B}) \rightarrow (\mathbb{B} \times \cdots \times \mathbb{B})$; the subscript k has no effect on execution, and is purely a naming convention that is convenient when describing unrolling. The *input* is assumed to change synchronously with the clock, and the *output* is determined by the combinational Boolean function $g : (\mathbb{B} \times \cdots \times \mathbb{B}) \times (\mathbb{B} \times \cdots \times \mathbb{B}) \rightarrow (\mathbb{B} \times \cdots \times \mathbb{B})$. Note that the internal state of the circuit, represented by *state*, is observable *only* through g.

Partial evaluation of this kind of circuit typically requires specialisation of f and g, but may also involve either partly or completely unrolling the `while` loop. State minimisation is not performed[3]. A single unrolling yields the program

```
while true
    state_{k+1} := f(state_k, input_1)
    output_1 := g(state_{k+1}, input_1)
    state_{k+2} := f(state_{k+1}, input_2)
    output_2 := g(state_{k+2}, input_2)
endwhile
```

which can be equivalently expressed as

```
while true
    state_{k+2} := f(f(state_k, input_1), input_2)
    output_1 := g(f(state_k, input_1), input_1)
    output_2 := g(f(f(state_k, input_1), input_2), input_2)
endwhile
```

[2]Conventionally, f would be drawn on the left of the state flip-flops. This (equivalent) form allows unrolling to be described more conveniently.

[3]In synchronous circuits, state minimisation is often undesirable – though it reduces flip flop count, the extra state decoding logic required often adversely affects maximum clock rates. As an extreme example, the performance advantages of a carefully designed one-hot encoded state machine might be lost entirely if state minimisation was to be naïvely attempted.

corresponding to the circuit shown in Fig. 2. After unrolling, since $state_{k+1}$ is not externally observable, it need not be explicitly computed. In one clock cycle, this new circuit performs the same computations that the original circuit performed in two cycles, though possibly with a slower maximum clock rate due to longer worst-case paths.

In the general case, since the $input_i$ and $output_j$ may change at every cycle, they may need to be separately accessible in the unrolled circuit. Often, though, *input* may be known to remain unchanged for many iterations, or for all time. It is common, also, for outputs other than those resulting from the final state to be unimportant; in combination, this allows loop unrolling to generate much more efficient hardware.

In the rest of this paper, we make the assumption that *input* may change only synchronously with the the clock of the generated hardware, and that *output* reflects the final state of the unrolled loop body at the end of each clock cycle.

3.1 Multiple Unrollings

Loops may be unrolled an arbitrary number of times by the following method:

```
while true
    state_{k+1} := f(state_k, input)
    state_{k+2} := f(state_{k+1}, input)
    ...
    state_{k+n} := f(state_{k+n-1}, input)
    output := g(state_{k+n}, input)
endwhile
```

or, equivalently:

```
while true
    state_{k+n} := f(f(...f(state_k, input), ... input)
    output := g(state_{k+n}, input)
endwhile
```

Since the repetition of *input* gives potential for common subexpression elimination, and g needs to be evaluated exactly once regardless of the number of unrollings (see Fig. 3), the gate count of any resulting circuit is typically (and often substantially) less than $n \times |f| + |g|$, where $|f|$ is the gate count of the original f, $|g|$ is the gate count of the original g, and n is the number of unrollings. In some cases, the final gate count may even be less than $|f| + |g|$ (see Section 4).

Figure 3: Synchronous Circuit After n Unrollings

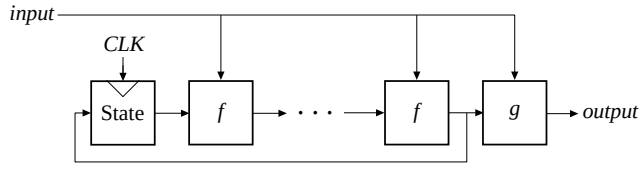
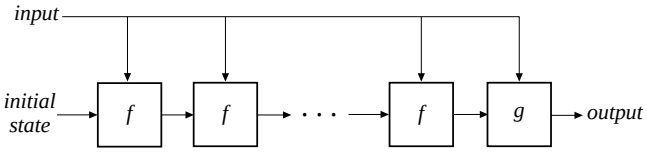


Figure 4: Synchronous Circuit After Full Unrolling



3.2 Reset Logic

Most synchronous circuits require some form of reset capability, corresponding to the following program:

$state_1 := initialstate$
while *true*
  $state_{k+1} := f(state_k, input)$
  $output := g(state_{k+1}, input)$
endwhile

In pure-synchronous hardware terms, since *state* may only change at a clock edge, it is not possible to have code execute outside the loop, so practical implementations usually resemble the following:

while *true*
  if $reset = true$
    $state_k := initialstate$
  endif
  $state_{k+1} := f(state_k, input)$
  $output := g(state_{k+1}, input)$
endwhile

Here, *reset* is a special synchronous input that causes *state* to be reset to *initialstate* if it is held *true* for one or more cycles. Note that, in this form, the circuit is just a special case of the general definition given in Section 3.

3.3 Full Unrolling

Where repeated application of f reaches a fixed point, i.e. where

$$state_0 = initialstate$$
$$state_{k+1} = f(state_k, input)$$

and there exists an n such that for all values of *input*, $state_{n+1} = state_n$, it is possible to fully unroll (and therefore eliminate) the **while** loop:

$state_0 := initialstate$
$state_1 := f(state_0, input)$
$state_2 := f(state_1, input)$
...
$state_n := f(state_{n-1}, input)$
$output := g(state_n, input)$

Any resulting circuit will be purely combinational (see Fig. 4); all D-type flip flops will have been eliminated.

4. THE HARPE LANGUAGE

HarPE (pronounced 'harpie') is a simple hardware description language created specifically to aid experimentation in partial evaluation. As is becoming increasingly common [4], HarPE is an *embedded* language, existing within a larger, more sophisticated general purpose programming language, in this case C++.

The HarPE language currently exists as an ISO C++ template library, taking advantage of template metaprogramming techniques [26, 27]. Compiling and then executing a C++ source file incorporating HarPE code causes a hardware netlist to be generated. The current compiler generates gate-level Verilog for further processing by a conventional tool chain.

4.1 Semantics

HarPE source code has a standard, imperative semantics, with the characteristic that a whole program defines *exactly one* machine cycle. An implicit outer *while* loop, executing once per clock cycle, is assumed, where one execution of the loop body corresponds to exactly one clock cycle[4] Partial evaluation is carried out aggressively as compilation proceeds.

As a simple example, the following program

```
Bit reset("reset");
IntReg<8> a;
a = a + 1;
If(reset);
    a = 0;
EndIf();
Output("a", a);
```

implements an 8-bit up counter with a synchronous reset input. The generated circuit outputs one count per clock cycle; though the source code has an imperative semantics, sequential composition does *not* mean that one or more clock cycles must take place – rather, in all cases, sequential composition requires *exactly zero* clock cycles.

4.2 Types

4.2.1 `Bit`

The fundamental type within HarPE is `Bit`, representing a single bit. Declarations follow C++ syntax:

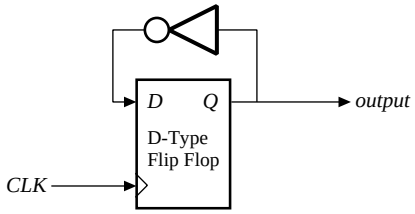
```
Bit a, b;
```

C++ operator overloading [21] is used to implement the logical operators representing *and*, *or* and *not*:

```
Bit a, b, c, d, e;
c = a & !b;
d = b | c;
e = a | d;
```

[4]This convention is also adopted by Verilog, though some other hardware description languages (notably Handel-C [5]) explicitly describe behaviour over many cycles.

Figure 5: A 1-bit 'Counter'



By default, `Bit` variables are initialised to *false*, so

```
Bit a;
Output("a", a);
```

will result in an output, labelled "a", that is connected directly to *false* (ground).

4.2.2 `BitReg`

D-type flip flops are represented by the `BitReg` type. `BitReg` behaves almost identically to `Bit`, with the exception that variables are initialised to reference a D-type flip flop. Any modifications to the value of a `BitReg` variable are incorporated into the feedback loop of the flip flop.

The circuit shown in Fig. 5 results from the following code:

```
BitReg a;
a = !a;
```

4.2.3 `Int`⟨n⟩

A variable of type `Int`⟨n⟩ represents a n-bit wide unsigned integer, implemented as an array of `Bit`s, with operator overloads supporting the usual arithmetic operators. Since `Bit`'s functionality is inherited, `Int`s are initialised to 0.

A number of alternative constructors are supported, including numeric constants, though they must be explicitly introduced (see the example in Section 4.1). For example, the following code generates an 8-bit multiplier:

```
Int<8> a, b, c;
c = a * b;
```

Individual bits within an `Int` may be addressed through the standard C++ array notation:

```
Int<8> a;
Bit b;
b = a[3];
a[4] = a[1];
```

If the subscript is a compile-time constant, HarPE simply provides access to the relevant underlying `Bit`. Where the subscript is itself an `Int`-valued expression, HarPE generates an appropriate multiplexer circuit.

4.2.4 `IntReg`⟨n⟩

`IntReg` is to `Int` what `BitReg` is to `Bit`; it allows multi-bit registers (normally representing unsigned integers) to be defined straightforwardly. As with `Int`, overloaded numeric operators support the usual arithmetic functions.

4.3 Inputs

Inputs are introduced by passing a parameter to the constructor of `Bit` or `Int`:

```
Bit x("x"), y("y"), z;
z = x | y;
```

In this example, z represents the output of an *or* gate whose inputs are the external inputs x and y.

Similar functionality is provided by `Int`:

```
Int<8> a("a"), b("b"), c;
c = a + b;
```

In this case, a pair of 8-bit input ports (named a[0..7] and b[0..7] in the netlist) are declared, with c representing the output of an 8-bit adder whose inputs are a and b.

4.4 Outputs

All outputs must be declared through the overloaded function `Output`(*"name"*, *expression*) , which can accept variables or expressions of type `Bit`, `BitReg`, `Int` or `IntReg`:

```
Bit x("x"), y("y"), z;
Int<8> a("a"), b("b"), c;
z = x | y;
c = a + b;
Output("z", z);
Output("c", c);
Output("q", x & y);
```

4.5 Compilation of Control Flow Constructs

The HarPE compiler flattens all control flow, so programs that do not require D-type flip flops (i.e. those programs that do not use variables of type `BitReg` or `IntReg`) always generate purely combinational hardware – such programs, in effect, execute in exactly zero clock cycles. When D-type flip flops are used, HarPE programs define what happens during exactly one clock cycle of the generated hardware. In this section, we describe how this is achieved.

4.5.1 Guarded Assignment

During compilation, HarPE maintains at all times a *guard expression*, Γ, that represents whether or not assignment statements should take place. At the start of compilation, $\Gamma = \mathit{true}$, so all assignments are valid. Control flow statements 'and' extra terms into Γ. The HarPE compiler maintains a closure stack, allowing block structured code with arbitrary nesting depth to be handled.

All assignment statements in HarPE, e.g.:

$$\mathit{var} = \mathit{newvalue};$$

are transformed internally to the following form:

$$\mathit{var}' = \begin{cases} \mathit{var} & \text{iff } \Gamma = \mathit{false}, \\ \mathit{newvalue} & \text{iff } \Gamma = \mathit{true} \end{cases}$$

All subsequent references to var in the program are renamed to var'. At bit level, this is equivalent to a simple multiplexer:

$$\mathit{var}' = (\Gamma \wedge \mathit{newvalue}) \vee (\neg\Gamma \wedge \mathit{var}).$$

Where $\Gamma = \mathit{true}$, this simplifies to $\mathit{var}' = \mathit{newvalue}$. If $\Gamma = \mathit{false}$, the assignment simplifies to $\mathit{var}' = \mathit{var}$, i.e. the

assignment has no effect. As a consequence, multiplexers are only generated when they are actually necessary.

Guarded assignment has close parallels with existing work on *static single assignment* (SSA) form [9], though since control flow is fully incorporated into assignments, there is no equivalent of SSA's Φ-functions (control flow merge points).

4.5.2 `If..Else..EndIf`

The `If..Else..EndIf` control structure introduces the result of a conditional expression to the guard of all statements within its scope, e.g.:

$[\![\Gamma]\!]$
`If`$(cond_1)$;
  $[\![\Gamma \wedge cond_1]\!]$
  `If`$(cond_2)$;
    $[\![\Gamma \wedge cond_1 \wedge cond_2]\!]$
  `Else`();
    $[\![\Gamma \wedge cond_1 \wedge \neg cond_2]\!]$
  `EndIf`();
  $[\![\Gamma \wedge cond_1]\!]$
`EndIf`();
$[\![\Gamma]\!]$

Note that, following the usual convention, the `Else` clause may be omitted.

In the following example, an `If` construct implements a reset circuit for a 3 element 'one hot' encoded shift register:

`BitReg` $a1, a2, a3$;
`Bit` $rst($"rst"$), x$;
`If`(rst);
  $a1 = 1$;
  $a2 = a3 = 0$;
`EndIf`();
$x = a3; a3 = a2; a2 = a1; a1 = x$;

HarPE flattens this into the equivalent of the following:

`BitReg` $a1, a2, a3$;
`Bit` $rst($"rst"$), x$;
$a1 = (rst \;\&\; 1) \mid (\neg rst \;\&\; a1)$;
$a2 = (rst \;\&\; 0) \mid (\neg rst \;\&\; a2)$;
$a3 = (rst \;\&\; 0) \mid (\neg rst \;\&\; a3)$;
$x = a3; a3 = a2; a2 = a1; a1 = x$;

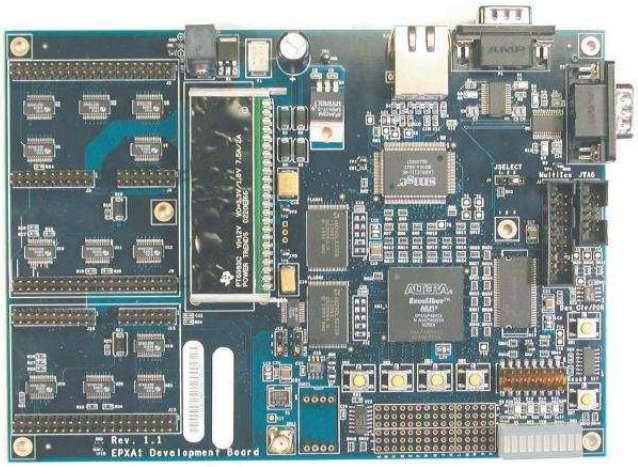
4.5.3 `While..EndWhile`

The HarPE `While..EndWhile` construct provides support for loop unrolling where the number of times the loop should execute may only be determined at run time, though a constant upper bound is necessary in order for compilation to terminate. The code sequence

`Int`$\langle 3\rangle$ $a(1), b(0), c($"stop"$)$;
`While`$(b < 3 \;\&\; !c[b])$;
  $a = a * a$
  $b = b + 1$
`EndWhile`();

loops through the bits of c, squaring the value of a each time as a side-effect, stopping either when the relevant bit of c is *true* or when the upper bound, 3, is reached. HarPE unrolls the loop equivalently to the following series of nested `If` statements:

Figure 6: Altera EPXA1 Development Board



`Int`$\langle 3\rangle$ $a(1), b(0), c($"stop"$)$;
`If`$(b < 3 \;\&\; !c[b])$;
  $a = a * a$
  $b = b + 1$
  `If`$(b < 3 \;\&\; !c[b])$;
    $a = a * a$
    $b = b + 1$
    `If`$(b < 3 \;\&\; !c[b])$;
      $a = a * a$
      $b = b + 1$
    `EndIf`();
  `EndIf`();
`EndIf`();

Unrolling terminates when the condition of the `While..EndWhile` loop can be determined to be *false* by combinational rewriting.

5. EXPERIMENTAL RESULTS

5.1 Test Environment and Experimental Procedures

For all of these experiments, code was compiled by HarPE, generating gate-level Verilog, which was then passed to Altera's Quartus II tool chain [2]. Each resulting circuit was compiled for an Altera Excalibur EPXA1F484C1 FPGA [1], then examined using the simulation tools within Quartus. Selected designs were uploaded to an Altera EPXA1 development board (see Fig. 6), though as this has a fixed 25MHz clock, timing information quoted below was calculated by post-layout timing simulation by the tool chain for designs that required a substantially different rate. The pure-combinational circuits were not characterised for timing.

5.1.1 *Empty Circuit*

To ensure that the gate count and other similar statistics were not skewed by something similar to the overhead of library code familiar in the software world, the following code

`Int`$\langle 7\rangle$ $c(0)$;
`Output`$($"R1"$, c)$;

was compiled and passed through the Quartus II tool chain. The test confirmed that, as expected, zero gates and zero flip flops were emitted by HarPE, resulting in a test FPGA which used zero logic elements (LEs).

5.2 Combinational PE

5.2.1 Specialising an Adder

The (unspecialised) program

```
Int⟨7⟩ a("a"), b("b");
Int⟨7⟩ c;
c = a + b;
Output("R1", c);
```

causes HarPE to emit 91 gates. Specialising b to the numeric value 1, i.e.

```
Int⟨7⟩ a("a"), b(1);
```

reduces the gate count to 36. In a cases where both a and b are specialised, e.g.:

```
Int⟨7⟩ a(25), b(9);
```

exactly zero gates are generated. Note that, as in all of these tests, HarPE performs partial evaluation *only* at bit level – it has no higher level rules dealing with integers or any other more complex data types.

Tying both inputs of the adder together:

```
c = a + a;
```

also results in a zero gate count, generating only wiring that performs a 'shift left' operation. Again, this results directly from bit level PE without higher level rules being necessary.

5.2.2 Specialising a Multiplier

Replacing $c = a + b$ in the test case shown in Section 5.2.1 with

```
c = a * b;
```

generates a 7bit × 7bit multiplier, emitting 443 gates. Specialising b to take the value 5 reduces this to just 58 gates.

The code

```
c = a * a;
```

generates a 'squarer' by tying the multiplier's inputs together. In this case the resulting gate count is 432, somewhat improved on the unspecialised version, though not so spectacularly as for addition.

Table 2: Loop Unrolling a 7-bit Up Counter

Loops per Cycle	Gates	DFFs	LEs	Max Clk
1	35	7	16	257MHz
2	69	7	13	257MHz
3	103	7	22	183.35Mhz
50	1701	7	18	257Mhz

Table 3: Loop Unrolling a Fibonacci Counter

Loops per Cycle	Gates	DFFs	LEs	Max Clk
1	107	14	38	163.03MHz
2	191	14	51	120.5MHz
3	275	14	52	96.83MHz
5	443	14	83	73.16MHz

5.3 Synchronous PE

5.3.1 Loop Unrolling of a Simple Counter

A simple, 7-bit up counter may be implemented as follows[5]:

```
IntReg⟨7⟩ reg;
reg = reg + 1;
Output("out", reg);
```

This circuit is particularly amenable to loop unrolling – see Table 2 for timing and gate count results. The disparity between the number of gates emitted by HarPE and the number of LEs generated by the Quartus II tool chain is indicative that the latter's more sophisticated combinational optimisation is successfully collapsing multiple increments into a single constant addition. Since HarPE emits purely bit-level Verilog, this optimisation must again be entirely bit-level in nature.

5.3.2 Loop Unrolling a Fibonacci Series Counter

The code

```
IntReg⟨7⟩ a, b;
Int⟨7⟩ temp;
Bit reset("rst");
If(reset);
    a = 1;
    b = 0;
EndIf();
temp = a + b;
b = a;
a = temp;
Output("out", a);
```

implements a specialised counter that outputs the Fibonacci series $(1, 2, 3, 5, 8, 13, 21, 34, \dots)$. These test cases, and those of Section 5.3.3, are loosely based on an example due to Page & Luk [20]. Test results are shown in Table 3. This time, maximum clock rate falls off as the number of unrollings increases – this is an expected (if not entirely welcome) feature of PE, and is caused by increasing propagation delays due to longer, more complex data paths.

[5]Note that there is an implicit outer *while* loop – see also Section 4.1

Table 4: Experimental Results for Partial Evaluation of a Small Processor

	Gates	DFFs	LEs	Max Clk	Run Time
Unmodified, 2 cycles per instruction	2029	75	646	27.48MHz	5.6μS (at 25MHz)
Merged fetch/execute, 1 instruction per cycle	1810	67	588	28.3MHz	2.8μS (at 25MHz)
2 × unrolled, 2 instructions per cycle	3883	67	1426	16.05MHz	2.37μS (at 15MHz)
4 × unrolled, 4 instructions per cycle	8029	67	2776	8.72MHz	2.1μS (at 8.33MHz)
Fully unrolled, 1 loop iteration per cycle	107	14	36	153.92MHz	70nS (at 150MHz)

Table 5: Instruction Set

Opcode	Mnemonic	Description
000	`SKIP`	Do nothing
001	`LDC`	$acc := operand$
010	`LDA`	$acc := mem[operand]$
011	`STA`	$mem[operand] := acc$
100	`ADDA`	$acc := acc + mem[operand]$
101	`JMP`	$ip := operand$
110	`STOP`	Halt

5.3.3 Partial Evaluation of a Small Processor

Loosely following [20] we define a small, 7-bit microprocessor with one 7-bit general purpose register, 8 bytes of RAM and 8 bytes of ROM. Both the RAM and ROM are mapped into a single 16 byte address space, with address 0..7 being RAM and 8..15 being ROM. The contents of address 13 (labelled R_1 in the assembler source below) are externally visible as a 7 bit output port for simulation and verification purposes.

Instructions are all single byte, with the 3 most significant bits representing an opcode and the 4 least significant bits representing a single operand. The supported instruction set is shown in Table 5.

In all tests shown here, the ROM contains the following program:

$R_1 = 13$
$R_2 = 14$
$X = 15$

start :	`LDA`	R_2
	`ADDA`	R_1
	`STA`	X
	`LDA`	R_2
	`STA`	R_1
	`LDA`	X
	`STA`	R_2
	`JMP`	*start*

A hardware reset circuit preinitialises R_2 with the value 1. All other locations are initialised to 0. Since the program loops forever unless externally terminated, run times were measured by layout aware timing simulation in Quartus II, measuring from the falling edge of the reset pulse to the time that R_1 reaches the arbitrarily chosen value 34 decimal (0100010 binary).

The basic processor was implemented in HarPE and instrumented to allow various levels of loop unrolling to be applied. Test results are shown in Table 4. Without unrolling, the processor requires 2029 gates (646 LEs), and executes one instruction every 2 clock cycles due to an explicit two phase fetch/execute cycle. Flattening this to one cycle, somewhat surprisingly, *reduces* the gate count and maintains a roughly similar maximum clock rate, halving the run time of the program[6]. Further unrolling generated versions of the processor that executed 2 and 4 instructions per clock cycle – simulation showed that these versions worked correctly, but increasing worst case propagation delays appeared to restrict the practical speedups that could be achieved.

Fully unrolling the loop so that the entire loop executes one iteration per clock cycle causes a dramatic reduction in gate count along with a large increase in speed. The resulting circuit compares well with the simple Fibonacci counter described in Section 5.3.2 – partial evaluation apparently optimises away the processor, leaving behind only the hardware necessary to implement the ROM program.

6. RELATED WORK

The first author's 1991 M.Sc thesis [22] described a hardware compiler based upon partial evaluation – this paper significantly extends that work and places it in a modern context. In other work [25], HarPE has been used to flatten circuits (in this case, small areas of an FPGA) to a combinational form suitable for analysis by a SAT solver.

The *Dynamic Synthesis of Correct Hardware* project [16, 15] at the University of Glasgow, which ran from May 1997 to May 1999, reported encouraging results from bit-level combinational PE, though did not address loop unrolling. The Bluespec hardware compiler [3] performs more extensive partial evaluation, though at an earlier compiler phase (i.e. not at bit-level).

7. CONCLUSIONS

The experimental results shown in Section 5 clearly show that partial evaluation of synchronous hardware is feasible. Partial loop unrolling offers designers an ability to specify circuits relatively simply, then transform them into faster (though possibly more complex) circuits purely by transformation. Full unrolling goes further, making it possible (as demonstrated in Section 5.3.3) to transform a processor and a ROM image into equivalent, low-level dedicated hardware – potentially, any synchronous soft core processor, in conjunction with a suitable partial evaluator, can be used as a hardware compiler for the machine language interpreted by the soft core itself.

In all of our tests, partial evaluation gave a net speed gain in comparison with the original circuit. In some cases,

[6]Though the reason for this reduction is unclear, it seems likely to be an artefact of our very simple fetch/execute implementation and is unlikely to be exhibited when specialising more complex processors.

gate count was also reduced. Full unrolling gave the most extreme results, with a 2 orders of magnitude speed up and 1 order of magnitude reduction in gate count.

7.1 Future Work

7.1.1 Automated Retiming/Pipelining

The timing information in Section 4 indicates that increasing worst-case path delays place a limit on the level of speedup that can be achieved with loop unrolling. Such circuits would almost certainly gain significantly in performance if they were pipelined and/or retimed [6], so it would be highly desirable to develop a technique that achieves this automatically, perhaps by bit-level transformation of the circuit. This would appear to be relatively straightforward for combinational circuits (and hence also any fully-unrolled synchronous circuit), but pipelining partially unrolled circuits appears to be non-trivial.

7.1.2 PE of Asynchronous Circuits

Performing PE of asynchronous circuits is fundamentally more difficult than the equivalent transformation of synchronous circuits. Rewrite rules that are perfectly safe when applied to synchronous circuits may alter the dynamic behaviour of asynchronous circuits [24, 23], introducing dangerous glitch states that could cause the circuit to function erratically, if at all. Restricting a partial evaluator only to known, safe, rewrite rules is one possible way forward which is likely to be suitable for specialisation, but no straightforward equivalent to loop unrolling appears to exist.

7.1.3 Abstract Interpretation

Abstract interpretation [7, 8] is often used in combination with PE, usually to determine whether or not it is appropriate to unroll loops. Applying similar techniques, such as representing values as convex polyhedra, may make it possible, for example, to optimise a soft core CPU against a particular program *without* performing full loop unrolling – the CPU's architecture could be retained, with hardware required for unused instructions optimised away.

7.1.4 Extending HarPE

The current implementation of HarPE supports experimental work (as in Section 5) quite well, but is not yet suitable for production quality hardware design. Extending its capabilities to better match the architecture and capabilities of contemporary target platforms (FPGAs, ASICs, etc.) would be required in order to make it suitable for commercial use.

7.1.5 PE of an Existing Soft Core

In Section 5.3.3, partial evaluation of a very simple microprocessor was demonstrated. Attempting a similar experiment based on an existing soft core CPU, rather than a purpose-built example, is a logical next step. Repeating the full unrolling experiment would be of particular interest.

Acknowledgements

The first author wishes to thank Big Hand Ltd., NASA, Intel, EPSRC and St Edmund's College, Cambridge for financially supporting this work. Experimental work relied heavily on software and equipment donated by Altera, for which grateful thanks are also due.

8. REFERENCES

[1] *Excalibur Device Overview Data Sheet, V2.0.* Altera, 2002. DS-EXCARM-2.0.

[2] *Quartus II Development Software Handbook, V4.0.* Altera, 2004.

[3] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification (Invited Talk),. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03)* (2003), p. 249.

[4] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. Lava: Hardware design in Haskell. In *International Conference on Functional Programming* (1998), ACM.

[5] Celoxica. Handel-C language reference manual. Available from http://www.celoxica.com/.

[6] Cong, J., and Wu, C. FPGA synthesis with retiming and pipelining for clock period minimization of sequential circuits. In *Proceedings of the 34th annual conference on Design automation conference* (1997), ACM Press, pp. 644–649.

[7] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, California, 1977), ACM Press, New York, NY, pp. 238–252.

[8] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, 1979), ACM Press, New York, NY, pp. 269–282.

[9] Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, F. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (1991), 451–490.

[10] Futamura, Y. Partial evaluation of computation process – an approach to a compiler-compiler. In *Systems, Computers, Control* (1971), vol. 2 issue 5, pp. 45–50.

[11] Hymans, C. Checking safety properties of behavioral VHDL descriptions by abstract interpretation. In *9th International Static Analysis Symposium (SAS'02)* (2002), vol. 2477 of *Lecture Notes in Computer Science*, Springer, pp. 444–460.

[12] Jones, N., Gomard, C., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[13] Lombardi, L. Incremental computation. In *Advances in Computers, vol. 8*, F. Alt and M. Rubinoff, Eds. New York: Academic Press, 1967, pp. 247–333.

[14] Lombardi, L., and Raphael, B. Lisp as the language for an incremental computer. In *The Programming Language Lisp: Its Operation and Applications* (1964), E. Berkeley and D. Bobrow, Eds., Cambridge, MA: MIT Press, pp. 204–219.

[15] McKay, N., Melham, T., Susanto, K. W., and Singh, S. Dynamic specialisation of XC6200 FPGAs

by partial evaluation. In *IEEE Symposium on FPGAs for Custom Computing Machines* (1998), K. L. Pocek and J. M. Arnold, Eds., IEEE Computer Society, pp. 308–309.

[16] McKay, N., and Singh, S. Dynamic specialisation of XC6200 FPGAs by partial evaluation. In *Field-Programmable Logic and Applications: From FPGAs to Computing Paradigm: 8th International Workshop, FPL'98, Estonia, 1998* (1998), R. W. Hartenstein and A. Keevallik, Eds., vol. 1482 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 298–307.

[17] Mealy, G. H. A method for synthesizing sequential circuits. In *Bell System Technical Journal* (1955), vol. 34, pp. 1045–1079.

[18] Mycroft, A., and Jones, N. D. A relational framework for abstract interpretation. In *Lecture Notes in Computer Science: Proc. Copenhagen workshop on programs as data objects* (1984), vol. 215, Springer-Verlag.

[19] Mycroft, A., and Sharp, R. W. Hardware synthesis using SAFL and application to processor design. In *Lecture Notes in Computer Science: Proc. CHARME'01* (2001), vol. 2144, Springer-Verlag.

[20] Page, I., and Luk, W. Compiling Occam into FPGAs. In *FPGAs*, W. Moore and W. Luk, Eds. Abingdon EE&CS Books, 1991, pp. 271–283.

[21] Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, USA, 1991.

[22] Thompson, S. Hardware compilation as an alternative computation architecture. Master's thesis, University of Teesside, 1991.

[23] Thompson, S., and Mycroft, A. Abstract interpretation of combinational asynchronous circuits. In *11th International Static Analysis Symposium (SAS'04)* (2004), R. Giacobazzi, Ed., vol. 3148 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 181–196.

[24] Thompson, S., and Mycroft, A. Sliding window logic simulation. In *15th UK Asynchronous Forum* (2004), Cambridge.

[25] Thompson, S., Mycroft, A., Brat, G., and Venet, A. Automatic in-flight repair of FPGA cosmic ray damage. In *Proc. 1st Disruption in Space Symposium* (July 2005).

[26] Veldhuizen, T. Using C++ template metaprograms. *C++ Report 7*, 4 (May 1995), 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.

[27] Veldhuizen, T. L. C++ templates as partial evaluation. In *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio* (Jan. 1999), University of Aarhus, Dept. of Computer Science, pp. 13–18.